Efficient Instantiation of Parameterised Boolean Equation Systems to Parity Games

Gijs Kant*
kant@cs.utwente.nl

Jaco van de Pol
vdpol@cs.utwente.nl

Formal Methods & Tools
University of Twente
Enschede, The Netherlands

Parameterised Boolean Equation Systems (PBESs) are sequences of Boolean fixed point equations with data variables, used for, e.g., verification of modal μ-calculus formulae for process algebraic specifications with data.

Solving a PBES is usually done by instantiation to a Parity Game and then solving the game. Practical game solvers exist, but the instantiation step is the bottleneck.

We enhance the instantiation in two steps. First, we transform the PBES to a Parameterised Parity Game (PPG), a PBES with each equation either conjunctive or disjunctive. Then we use LTSMIN, that offers transition caching, efficient storage of states and both distributed and symbolic state space generation, for generating the game graph. To that end we define a language module for LTSMIN, consisting of an encoding of variables with parameters into state vectors, a grouped transition relation and a dependency matrix to indicate the dependencies between parts of the state vector and transition groups.

Benchmarks on some large case studies, show that the method speeds up the instantiation significantly and decreases memory usage drastically.

Key words: Parameterised Boolean Equation Systems, Parity Games, Instantiation, LTSMIN.

1 Introduction

Parameterised Boolean Equation Systems (PBESs) are sequences of fixed point equations with data variables. They form a very expressive formalism for encoding a wide range of problems, such as the verification of modal μ-calculus formulae [14, 6] for process algebraic specifications with data (see, e.g., [10, 11]) and checking for (branching) bisimilarity of process equations [7].

PBESs have been described extensively in [11]. A method for solving PBESs directly has been presented [10], but usually PBESs are solved by first instantiating the system to a plain Boolean Equation System (BES) and then solving the BES. Instantiation of PBESs is described in [8, 16], where clever rewriters and enumeration of quantifier expressions play an important role. We focus on instantiation to a Parity Game (PG), which is a restricted BES with equations that are either conjunctive or disjunctive. Although no polynomial time algorithm for solving parity games is known (however, the problem is known to be in NP $\cap$ co-NP), effective parity game solvers exist (see, e.g., [9]), especially when the alternation depth is low, and the instantiation step is currently the bottleneck of the whole procedure for many practical cases.

*Gijs Kant is sponsored by the NWO under grant number 612.000.937 (VOCHS).

A.J. Wijs, D. Bošnački, S. Edelkamp (Eds.): GRAPH
Inspection and Traversal Engineering (GRAPHITE 2012)
EPTCS 99, 2012, pp. 50–65, doi:10.4204/EPTCS.99.7

There are clear similarities between instantiation of PBESs and state space generation, a well known problem in model checking. In both, an abstract description gives rise to a large graph, which requires efficient storage of the generated graph. Also, in both we often have that the description consists of a combination of reasonably independent components or equations. This 'locality' can be used to speed up the generation of successor nodes. Inspired by these similarities, we apply in this paper optimisations from model checking to the PBES instantiation problem, devising a more efficient method. We use LTSMIN, a language independent toolset for state space exploration which enables efficient state space generation and offers both symbolic exploration tools based on Binary Decision Diagrams (BDDs) and distributed exploration tools (see, e.g., [5]). The tools make use of knowledge about the dependencies for better efficiency, which can be specified for every language in a separate language module. Instantiating PBESs to parity games in our enhanced method has two phases:

1) Transforming the PBES into an equivalent system that consists of expressions that are either purely conjunctive or purely disjunctive. We call such a system a *Parameterised Parity Game* (PPG). The result of this operation is that any instantiation of the PPG will result directly in a parity game.
2) Instantiating the PPG to a PG using LTSMIN. To this end this we defined a PBES language module for LTSMIN, in which we specify a state vector representation of instantiated PBES variables (and the corresponding node in the generated game graph) and the dependencies between (parts of) the equations and the parts of the state vector.

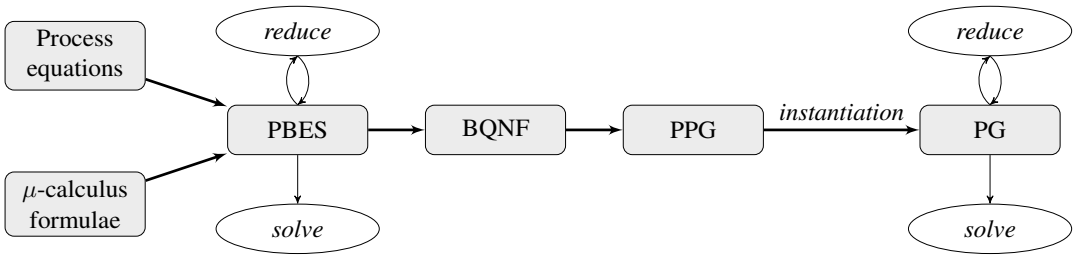

Figure 1: Overview of the verification approach, consisting of various transformations, an instantiation step, and available reductions and solvers.

An overview of the method is shown in Figure 1. We consider PBESs in *Bounded Quantifier Normal Form* (BQNF), which is a subset of all PBESs, but any PBES can be rewritten automatically to a system in BQNF with the same solution. PBESs and their normal forms are described in Section 2. The contributions of this article are the transformation from BQNF to PPG and the instantiation from PPG to PG. Both steps are not trivial. We will explain here where the obstacles lie.

In general, each system of PBES equations in BQNF can be transformed automatically into a system consisting of equations in PPG while preserving the solution. An equation can be transformed to PPG by introducing fresh equations for subexpressions and replacing the subexpressions by the corresponding variable. However, it is important not to separate quantifiers from the expressions that restrict the data elements that have to be considered, so called *bounds*. If a bound for a quantifier over an infinite data sort is replaced by a variable, the instantiator might generate an infinite number of successors for a node in the game graph. See Section 3 for our solution.

For the instantiation step we implemented a PBES language module for LTSMIN using the Partitioned Interface for the Next State function (PINS). This includes partitioning each PBES equation into *transition groups* and defining a *dependency matrix* that specifies the dependencies between transition group and parts of the state vector. We then have a high-performance instantiation tool that offers both distributed

and symbolic generation of a parity game. This requires some delicacy, as splitting a formula too much may result in infinite computation (as in the transformation phase) and not splitting enough could result in a dependency matrix that is too dense, which ruins the effect of transition caching and symbolic computation. The implementation is described in Section 4.

In Section 5 we present performance results for a number of case studies, comparing our sequential, distributed and symbolic implementations based on LTSMIN to the existing PBES instantiation tools in the mCRL2 toolset. In almost all cases memory usage is orders of magnitude better for our tool. In all cases also the execution time is much better.

2 Background

In this section we will treat PBESs, normal forms for PBESs, and Parity Games.

2.1 PBES

Definition 2.1. *Predicate formulae* φ are defined by the following grammar:

$$\varphi ::= b \mid \mathsf{X}(\vec{e}) \mid \neg\varphi \mid \varphi \oplus \varphi \mid \mathsf{Q}d : D \,.\, \varphi$$

where $\oplus \in \{\wedge, \vee, \Rightarrow\}$, $\mathsf{Q} \in \{\forall, \exists\}$, b is a data term of sort Bool, $\mathsf{X} \in \mathcal{X}$ is a predicate variable, d is a data variable of sort D, and $\vec{e}$ is a vector of data terms. We will call any predicate formula without predicate variables a *simple formula*. We denote the class of predicate formulae $\mathcal{F}$.

Definition 2.2. A *First-Order Boolean Equation* is an equation of the form:

$$\sigma\mathsf{X}(\vec{d} : D) = \varphi$$

where $\sigma \in \{\mu, \nu\}$ is a minimum (μ) or maximum (ν) fixed point operator, $\vec{d}$ is a vector of data variables of sort D, and φ is a predicate formula.

Definition 2.3. A *Parameterised Boolean Equation System (PBES)* is a sequence of First-Order Boolean Equations:

$$\mathcal{E} = (\sigma_1\mathsf{X}_1(\vec{d_1} : D_1) = \varphi_1) \;\; \ldots \;\; (\sigma_n\mathsf{X}_n(\vec{d_n} : D_n) = \varphi_n)$$

The semantics and solution of PBESs are described in, e.g., [11]. We say that two equation systems $\mathcal{E}_1$ and $\mathcal{E}_2$ are equivalent, written as $\mathcal{E}_1 \equiv \mathcal{E}_2$, if they have the same solution for every variable that occurs in both systems.

We adopt the standard limitations: expressions are in positive form (negation occurs only in data expressions) and every predicate variable occurs exactly once as the left hand side of an equation. A PBES that contains no quantifiers and parameters is called a *Boolean Equation System* (BES). A finitary PBES can be *instantiated* to a BES by expanding the quantifiers to finite conjunctions or disjunctions and substituting concrete values for the data parameters. Every instantiated PBES variable $\mathsf{X}(\vec{e})$ should then be read as a BES variable "$\mathsf{X}(\vec{e})$".

A one-to-one mapping can be made from a BES to an equivalent *parity game* if the BES has only expressions that are either conjunctive or disjunctive. The parity game is then represented by a game graph with nodes that represent variables with concrete parameters and edges that represent dependencies. Parity games will be further explained in Section 2.2. To make instantiation of a PBES to a parity game more directly we will preprocess the PBES to a format that only allows expressions to be either conjunctive or disjunctive. This format is a normal form for PBESs that we call the *Parameterised Parity Game*, defined as follows:

Definition 2.4. A PBES is a *Parameterised Parity Game* (PPG) if every right hand side of an equation is a formula of the form:

$$\bigwedge_{i\in I} f_i \wedge \bigwedge_{j\in J} \forall_{\vec{v}\in D_j} \,.\, \big(g_j \Rightarrow \mathsf{X}_j(\vec{e_j})\big) \quad | \quad \bigvee_{i\in I} f_i \vee \bigvee_{j\in J} \exists_{\vec{v}\in D_j} \,.\, \big(g_j \wedge \mathsf{X}_j(\vec{e_j})\big).$$

where f_i and g_j are simple boolean formulae, and $\vec{e_j}$ is a data expression. I and J are finite (possibly empty) index sets.

The expressions range over two index sets I and J. The left part is a conjunction (or disjunction) of simple expressions f_i that can be seen as conditions that should hold in the current state. The right part is a conjunction (or disjunction) of a quantified vector of variables for next states X_j with parameters $\vec{e_j}$, guarded by simple expression g_j.

Before transforming arbitrary PBESs to PPGs we first define another normal form on PBESs to make the transformation easier. This normal form can have an arbitrary sequence of bounded quantifiers as outermost operators and has a conjunctive normal form at the inner. We call this the Bounded Quantifier Normal Form (BQNF):

Definition 2.5. A First-Order Boolean formula is in *Bounded Quantifier Normal Form (BQNF)* if it has the form:

$$\begin{aligned}
\mathsf{BQNF} &::= \quad \forall \vec{d} \in D \,.\, b \Rightarrow \mathsf{BQNF} \quad | \quad \exists \vec{d} \in D \,.\, b \wedge \mathsf{BQNF} \quad | \quad \mathsf{CONJ} \\
\mathsf{CONJ} &::= \quad \bigwedge_{k\in K} f_k \wedge \bigwedge_{i\in I} \forall_{\vec{v}\in D_I} \,.\, \big(g_i \Rightarrow \mathsf{DISJ}^i\big) \\
\mathsf{DISJ}^i &::= \quad \bigvee_{\ell\in L_i} f_{i\ell} \vee \bigvee_{j\in J_i} \exists_{\vec{w}\in D_{ij}} \,.\, \big(g_{ij} \wedge \mathsf{X}_{ij}(\vec{e_{ij}})\big)
\end{aligned}$$

where b, f_k, $f_{i\ell}$, g_i, and g_{ij} are simple boolean formulae, and $\vec{e_{ij}}$ is a data expression. K, I, L_i, and J_i are finite (possibly empty) index sets.

This BQNF is similar to *Predicate Formula Normal Form* (PFNF), defined elsewhere[1], in that quantification is outermost and in that the core is a conjunctive normal form. However, unlike PFNF, BQNF allows bounds on the quantified variables (hence bounded quantifiers), and universal quantification is allowed within the conjunctive part and existential quantification is allowed within the disjunctive parts. These bounds are needed to avoid problems when transforming to PPG. Consider the expression $(\forall i : \mathbb{N} \,.\, (i < 5) \Rightarrow \mathsf{Y}(i)) \vee (\exists j : \mathbb{N} \,.\, (j < 3) \wedge \mathsf{Z}(j))$. Rewriting to PFNF (moving the quantifiers outward) results in $\exists j : \mathbb{N} \,.\, \forall i : \mathbb{N} \,.\, ((i < 5) \Rightarrow Y(i)) \vee ((j < 3) \Rightarrow Z(j))$. Rewriting that expression to PPG would split the expression such that the initial expression is $\exists j : \mathbb{N} \,.\, \mathsf{X}_1(j)$ (X_1 is a newly introduced variable for the equation with the remainder of the expression as right hand side), which would result in an infinite disjunction when instantiating the PPG. BQNF allows the original expression to be rewritten to $\exists j : \mathbb{N} \,.\, (j < 3) \wedge \forall i : \mathbb{N} \,.\, (i < 5) \Rightarrow (\mathsf{Y}(i) \vee \mathsf{Z}(j))$ with the bounds close to the quantifiers, which allows to split the expression after the bound, preventing the instantiation to result in an infinite expression. Requiring that a system is specified in BQNF does not limit the expressiveness, as each PBES can be transformed into a equivalent system in PFNF that has the same solution and PFNF is a subset of BQNF.

The translation from process algebraic specifications in mCRL2 and μ-calculus formulae to PBESs is given in [10] and is illustrated by the following example. Throughout the paper we expect the reader to know process algebras and to be able to read mCRL2 specifications[2].

[1] A transformation to PFNF is implemented in the `pbesrewr` tool and documented at `http://www.win.tue.nl/mcrl2/wiki/index.php/Parameterised_Boolean_Equation_Systems`.

[2] See `http://mcrl2.org` for documentation on the mCRL2 language.

Example 2.6 (Buffer)**.** Consider the specification of a simple buffer with a capacity of 2.

$$\begin{aligned}
&\textbf{sort}\ \ D = \textbf{struct}\ d_1 \mid d_2; \quad \textbf{act}\ \mathrm{r}_1, \mathrm{s}_4 : D;\\
&\textbf{proc}\ \mathsf{Buffer}(q : \mathrm{List}(D)) = \sum_{d:D} (\#q < 2) \to \mathrm{r}_1(d)\ .\ \mathsf{Buffer}(q \triangleleft d)\\
&\qquad\qquad + (q \neq []) \to \mathrm{s}_4(\mathrm{head}(q))\ .\ \mathsf{Buffer}(\mathrm{tail}(q));\\
&\textbf{init}\ \ \mathsf{Buffer}([]);
\end{aligned}$$

The specification consists of sort and action definitions, process specifications where alternatives are in summands and an initial state. On the first line an enumerated data sort D is introduced with data values d_1 and d_2, and the actions r_1 and s_4 are specified, both having a data parameter of type D. A process Buffer is specified that has a data parameter q, which is a list of elements of type D. The process consists of *summands*, separated by the $+$-operator. Each summand may start with a summation over a data set, followed by a guard that is closed with a $\to$, then an action, followed by a call to the process that describes the behaviour after the action, typically a recursive call to the process itself with different parameters.

The first summand specifies that any element d can be added to q by the action $\mathrm{r}_1(d)$ if the size of the internal buffer q is smaller than 2. The second summand specifies that if q is not empty, elements can be popped by the action $\mathrm{s}_4(\mathrm{head}(q))$. The initial state of the system is the Buffer process with an empty list in this case, which models that initially the buffer is empty.

We can check the specification for absence of deadlock, which is expressed in μ-calculus as follows:

$$[\top^*]\langle\top\rangle\top \quad \text{(which is syntactic sugar for: } \nu X\ .\ \langle\top\rangle\top \wedge [\top]X)$$

which reads: after any sequence of actions ($[\top^*]$), always some action is enabled ($\langle\top\rangle\top$). Satisfaction of the formula by the specification, translated to a PBES, looks as follows:

$$\begin{aligned}
&\textbf{sort}\ \ D = \textbf{struct}\ d_1 \mid d_2;\\
&\textbf{pbes}\ \nu\mathsf{X}(q : \mathrm{List}(D)) = (q \neq []) \vee (\#q < 2)\\
&\qquad\qquad \wedge (q \neq []) \Rightarrow \mathsf{X}(\mathrm{tail}(q))\\
&\qquad\qquad \wedge \forall_{d\in D}\ .\ (\#q < 2) \Rightarrow \mathsf{X}(q \triangleleft d);\\
&\textbf{init}\ \ \mathsf{X}([]);
\end{aligned}$$

This PBES is true if from the initial state $\mathsf{X}([])$ an element can be added to q if $\#q$ is smaller than 2, an element can be popped from q if it is not empty and any of these actions is enabled ($q \neq []$ or $\#q < 2$, which is obviously true for any q). The same has to hold for the successor states (X with an element added to, respectively popped from q as parameter). The solution of the PBES is **true**.

Remark. The equation system in the example above is already a PPG, which is no coincidence as any system when combined with the absence of deadlock property will result in a PBES in PPG form because of the form of the formula: a conjunction of "we can do an action now" (a disjunctive expression without recursion) and "for all possible actions the property holds in all next states" (universal quantification with recursion). Note that checking the absence of deadlock property is almost the same as standard reachability analysis.

Definition 2.7 (Block)**.** A PBES is divided into *blocks*, which are subsequences of equations with the same fixed point operator such that subsequent equations with the same fixed point operator belong to the same block.

2.2 Parity Games

A *parity game* is a game between two players, player **0** (also called Eloise or player *even*) and player **1** (also called Abelard or player *odd*), where each player owns a set of places. On one place a token is

placed that can be moved by the owner of the place to an adjacent place. The parity game is represented as a graph. We borrow notation from [6] and [15].

Definition 2.8 (Parity Game)**.** A *parity game* is a graph $\mathcal{G} = \langle V, E, V_{\mathbf{0}}, V_{\mathbf{1}}, v_I, \Omega \rangle$, with

- V the set of vertices (or places or states);
- $E : V \times V$ the set of transitions;
- $V_{\mathbf{0}} \subseteq V$ the set of places owned by player **0**;
- $V_{\mathbf{1}} \subseteq V$ the set of places owned by player **1**;
- $v_I \in V$ the initial state of the game;
- $\Omega : V \to \mathbb{N}$ assigns a priority $\Omega(v)$ to each vertex $v \in V$;

where $V_{\mathbf{0}} \cup V_{\mathbf{1}} = V$ and $V_{\mathbf{0}} \cap V_{\mathbf{1}} = \emptyset$.

The nodes in the graph represent the places and correspond to the instantiated variables from the equation system. The edges represent possible moves of the token (initially placed on v_I) and encode dependencies between variables. A node does not necessarily have outgoing transitions, i.e., deadlock nodes are allowed. In the parity game, player **0** owns the nodes that represent disjunctions, player **1** the nodes that represent conjunctions.

The node priorities correspond to the number of the block to which the corresponding variable belongs (see Def. 2.7), such that variables in earlier blocks have lower priorities, ν-blocks have even priorities, μ-blocks have odd priorities and the earliest μ-block has priority 1. The following table shows an intuitive overview of the relations between BESs and parity games.

ν blocks	Even priorities (0, 2, 4, ...)
μ blocks	Odd priorities (1, 3, 5, ...)
$\vee$, $\exists$, $\langle\rangle$	Player **0**, $\exists$loise, Even, Prover
$\wedge$, $\forall$, $[]$	Player **1**, $\forall$belard, Odd, Refuter

The values **true** ($\top$) and **false** ($\bot$) are represented as a node with priority 0, player **1** and a transition to itself, and a node with priority 1, player **0** and a transition to itself, respectively.

A *play* in the game is a finite path $\pi = v_0 v_1 \cdots v_r \in V^+$ ending in a deadlock state v_r or an infinite path $\pi = v_0 v_1 \cdots \in V^\omega$ such that $(v_i, v_{i+1}) \in E$ for every $v_i \in \pi$. Priority function Ω extends to plays in the following way: $\Omega(\pi) = \Omega(v_0)\Omega(v_1)\cdots$. $\mathrm{Inf}(\rho)$ returns the set of values that occur infinitely often in a sequence ρ.

Definition 2.9 (Winner of a play)**.** Player **0** is the winner of a play π if

- π is a finite play $v_0 v_1 \cdots v_r \in V^+$ and $v_r \in V_{\mathbf{1}}$ and no move is possible from v_r; or
- π is an infinite play and $\min(\mathrm{Inf}(\Omega(\pi)))$, the minimum of the priorities that occur infinitely often in π, is even. This is called the *min-parity condition*.

Definition 2.10 (Strategy)**.** A (memoryless) *strategy* for player a is a function $f_a : V_a \to V$. A play $\pi = v_0 v_1 \cdots$ is *conform* to f_a if for every $v_i \in \pi$, $\quad v_i \in V_a \Rightarrow v_{i+1} = f_a(v_i)$.

Definition 2.11 (Winner of the game)**.** Player **0** is the *winner* of the game if and only if there exists a winning strategy for player **0**, i.e., from the initial state every play conforming to the strategy will be won by player **0**.

The model checking problem is encoded as a PBES (see [10]) which is instantiated to a parity game (see [16]) such that player **0** is the winner of the game iff the property holds for the system.

Solving Parity Games Solving a parity game means finding a winning strategy for one of the players. Various algorithms exist, such as the recursive algorithm by Zielonka [20] and Small Progress Measures by Jurdziński [13], with a multi-core implementation in [18]. An overview and performance comparison of the algorithms are given in [9].

3 Transformation from BQNF to Parameterised Parity Games

In order to automatically transform a PBES to a PPG, we define a transformation function s from BQNF to PPG. The transformation rewrites expressions that contain both conjunctions and disjunctions to equivalent expressions that are either conjunctive or disjunctive, by introducing new equations for certain subformulae and substituting calls to the new equations for these subformulae in the original expression. The function t below replaces an expression by a call to a new equation if the expression is not already a variable instantiation. The function t' introduces a new equation for an expression if needed.

$$t(\mathsf{X},\vec{d},\varphi) \stackrel{\text{def}}{=} \begin{cases} \varphi & \text{if } \varphi \text{ is of the form } \mathsf{X}'(\vec{e}), \\ \mathsf{X}(\vec{d}) & \text{otherwise;} \end{cases}$$

$$t'(\sigma,\mathsf{X},\vec{d},\varphi) \stackrel{\text{def}}{=} \begin{cases} \varnothing & \text{if } \varphi \text{ is of the form } \mathsf{X}'(\vec{e}), \\ s(\sigma\mathsf{X}(\vec{d}) = \varphi) & \text{otherwise.} \end{cases}$$

For brevity, we leave out the types of the parameters. A tilde is used to introduce a fresh variable: $\widetilde{\mathsf{X}}$. For equation system $\mathcal{E} = (\sigma\mathsf{X}_1(\vec{d_1}) = \xi_1) \ \ldots \ (\sigma\mathsf{X}_n(\vec{d_n}) = \xi_n)$, with each ξ_i in BQNF, the translation to PPG is defined as follows:

$$s(\mathcal{E}) \stackrel{\text{def}}{=} s\big(\sigma\mathsf{X}_1(\vec{d_1}) = \xi_1\big) \ \ldots \ s\big(\sigma\mathsf{X}_n(\vec{d_n}) = \xi_n\big)$$

$$s\big(\sigma\mathsf{X}(\vec{d}) = f\big) \stackrel{\text{def}}{=} \sigma\mathsf{X}(\vec{d}) = f$$

$$s\big(\sigma\mathsf{X}(\vec{d}) = \forall\vec{v} \,.\, b \Rightarrow \varphi\big) \stackrel{\text{def}}{=} \Big(\sigma\mathsf{X}(\vec{d}) = \forall\vec{v} \,.\, b \Rightarrow t(\widetilde{\mathsf{X}},\vec{d}+\vec{v},\varphi)\Big) \ t'(\sigma,\widetilde{\mathsf{X}},\vec{d}+\vec{v},\varphi)$$

$$s\big(\sigma\mathsf{X}(\vec{d}) = \exists\vec{v} \,.\, b \wedge \varphi\big) \stackrel{\text{def}}{=} \Big(\sigma\mathsf{X}(\vec{d}) = \exists\vec{v} \,.\, b \wedge t(\widetilde{\mathsf{X}},\vec{d}+\vec{v},\varphi)\Big) \ t'(\sigma,\widetilde{\mathsf{X}},\vec{d}+\vec{v},\varphi)$$

$$s\big(\sigma\mathsf{X}(\vec{d}) = \textstyle\bigwedge_{k\in K} f_k \wedge \bigwedge_{i\in I}(\forall_{\vec{v}_i} \,.\, g_i \Rightarrow \varphi_i)\big) \stackrel{\text{def}}{=} \Big(\sigma\mathsf{X}(\vec{d}) = \bigwedge_{k\in K} f_k \wedge \bigwedge_{i\in I}\big(\forall_{\vec{v}_i} \,.\, g_i \Rightarrow t(\widetilde{\mathsf{X}}_i,\vec{d}+\vec{v}_i,\varphi_i)\big)\Big) \ t'(\sigma,\widetilde{\mathsf{X}}_1,\vec{d}+\vec{v}_1,\varphi_1) \ \ldots \ t'(\sigma,\widetilde{\mathsf{X}}_m,\vec{d}+\vec{v}_m,\varphi_m)$$

$$s\big(\sigma\mathsf{X}(\vec{d}) = \textstyle\bigvee_{k\in K} f_k \vee \bigvee_{i\in I}(\exists_{\vec{v}_i} \,.\, g_i \wedge \varphi_i)\big) \stackrel{\text{def}}{=} \Big(\sigma\mathsf{X}(\vec{d}) = \bigvee_{k\in K} f_k \vee \bigvee_{i\in I}\big(\exists_{\vec{v}_i} \,.\, g_i \wedge t(\widetilde{\mathsf{X}}_i,\vec{d}+\vec{v}_i,\varphi_i)\big)\Big) \ t'(\sigma,\widetilde{\mathsf{X}}_1,\vec{d}+\vec{v}_1,\varphi_1) \ \ldots \ t'(\sigma,\widetilde{\mathsf{X}}_m,\vec{d}+\vec{v}_m,\varphi_m)$$

with $I = 1 \ldots m$, $\vec{v} \cap \vec{d} = \emptyset$ (variables in $\vec{v}$ do not occur in $\vec{d}$), b, f, f_k, g_i are simple formulae, φ, φ_i are formulae that may contain predicate variables.

Proposition 3.1. *The transformation s is solution preserving, i.e., for any $\mathcal{E}$ in BQNF, $s(\mathcal{E}) \equiv \mathcal{E}$: bound variables $\mathsf{X}(d)$ have the same solution in $s(\mathcal{E})$ as in $\mathcal{E}$.*

Proof. Every change made by s to an equation $\sigma\mathsf{X} = \xi$ is a substitution of a subexpression φ by a fresh variable $\widetilde{\mathsf{X}}$, while adding at the same time a new equation $\sigma\widetilde{\mathsf{X}} = \varphi$ in the same block as X. We can apply *backward substitution* (using [11, Lemma 18]) $s(\sigma\mathsf{X} = \xi)[\widetilde{\mathsf{X}} := \varphi]$ for every substitution caused by the transformation to get the original equation system (plus an unused equation $s(\sigma\widetilde{\mathsf{X}} = \varphi)$ for every fresh variable $\widetilde{\mathsf{X}}$). From that we can conclude that $s(\mathcal{E}) \equiv \mathcal{E}$. □

Example 3.2 (Example of the transformation)**.** We combine the buffer from Example 2.6 with the property that in every state both r_1 and s_4 actions are enabled:

$$\nu\mathsf{X} \,.\, (\exists_{d:D} \,.\, \langle \mathrm{r}_1(d) \rangle \mathsf{X}) \wedge (\exists_{d:D} \,.\, \langle \mathrm{s}_4(d) \rangle \mathsf{X})$$

The resulting PBES has an equation which does not conform to the PPG form, but is in BQNF:

$$\begin{array}{ll} \textbf{sort} & D = \textbf{struct}\ d_1 \mid d_2; \\ \textbf{pbes} & \nu\mathsf{X}(q : \mathrm{List}(D)) = \big(\exists_{d:D} \,.\, (\#q < 2) \wedge \mathsf{X}(q \triangleleft d)\big) \\ & \qquad\qquad \wedge \big(\exists_{d:D} \,.\, (\mathrm{head}(q) = d) \wedge (q \neq []) \wedge \mathsf{X}(\mathrm{tail}(q))\big); \\ \textbf{init} & \mathsf{X}([]); \end{array}$$

The transformation s replaces both conjuncts by a fresh variable and adds equations for these variables with the substituted expression as right hand side, resulting in equations:

$$\begin{array}{ll} \textbf{pbes} & \nu\mathsf{X}(q : \mathrm{List}(D)) \;\; = \mathsf{X}_1(q) \wedge \mathsf{X}_2(q); \\ & \nu\mathsf{X}_1(q : \mathrm{List}(D)) = \exists_{d:D} \,.\, (\#q < 2) \wedge \mathsf{X}(q \triangleleft d); \\ & \nu\mathsf{X}_2(q : \mathrm{List}(D)) = \exists_{d:D} \,.\, (\mathrm{head}(q) = d) \wedge (q \neq []) \wedge \mathsf{X}(\mathrm{tail}(q)); \end{array}$$

The first equation is purely conjunctive, while that latter two equations are (guarded) disjunctive.

4 Instantiation of Parameterised Parity Games

We view the instantiation of PPGs to Parity Games as generating a transition system, where states are predicate variables with concrete parameters and transitions are dependencies, specified by the right hand side of the corresponding equation in the PPG.

Example 4.1. Consider the equation:

$$\nu\mathsf{X}(d : D) = (d > 0 \wedge d < 10) \Rightarrow \mathsf{X}(d-1) \wedge \mathsf{X}(d+1)$$

If $\mathsf{X}(5)$ is the initial value, its successors are $\mathsf{X}(4)$ and $\mathsf{X}(6)$, so the graph starts with a node owned by player **1** representing $\mathsf{X}(5)$ with transitions to nodes $\mathsf{X}(4)$ and $\mathsf{X}(6)$.

4.1 LTSMIN

We use the tool LTSMIN to generate a parity game given a PPG. LTSMIN is a language independent tool for state-space generation [5]. Different language-modules are available, which are connected to different exploration algorithms through the so-called PINS-interface. This interface allows for certain language-independent optimisations, such as transition caching and distributed generation (see [4]), and an efficient compressed storage of states in a tree database (see [2]). Also symbolic reachability analysis is possible, where the state space is stored as a Binary Decision Diagram (BDD) [3].

4.1.1 Partitioned Interface for the Next State function

LTSMIN uses a Partitioned Interface for the Next State function (PINS), where states are represented as a vector $\langle x_1, x_2, \ldots, x_M \rangle$ with size M that is fixed for the whole system (to be determined statically). These values are stored in a globally accessible table, so that the states can also be represented as a vector of integer indices $\langle i_1, i_2, \ldots, i_M \rangle$. The PINS interface functions on this level of integer vectors, so that each tool can really be language-independent. Throughout the text we will often use value vectors instead of index vectors for better readability.

For a system with a state vector of M parts, the universe of states is $S = \mathbb{N}^M$. For each language module a *transition function* $\text{NEXT} : S \to \mathcal{P}(S)$ has to be defined that computes the set of successor states for a given state. This transition relation is preferrably split into *transition groups* in order to reflect the compositional structure of the system, by defining a function $\text{GROUP-NEXT} : S \times \mathbb{N} \to \mathcal{P}(S)$ that computes successors for state s as defined in group k. Suppose we have K transition groups. NEXT can then be defined as

$$\text{NEXT}(s) = \bigcup_{k=1}^{K} \text{GROUP-NEXT}(s, k)$$

4.1.2 Dependence

An important optimisation comes from the observation that not all parts of the state vectors are relevant in every transition group. To indicate the relevant parts of the vector for each of the transition groups, LTSMIN uses a *dependency matrix*, which has to be computed statically.

Definition 4.2 (PINS Matrix: [4], Def. 4)**.** A *dependency matrix* $D_{K \times N} = DM(P)$ for system P is a matrix with K rows and N columns containing $\{0, 1\}$ such that if $D_{k,i} = 0$ then group k is independent of element i.

For any transition group $1 \leq k \leq K$, we define π_k as the projection $\pi_k : S \to \Pi_{\{1 \leq i \leq N | D_{k,i} = 1\}} S_i$.

Independence here means that for given transition group k the transitions do not depend on part i of the state vector (*read independence*) and the transitions do not change part i of the successor state vector (*write independence*) or that part i is *irrelevant* in both the current state and all successor states. *Irrelevant* here means that changing the value of that part would still result in a bisimilar state space. For a more precise definition, see [17, Def. 9]. This definition of independence is slightly more liberal than the one in [4] in that we added this notion of relevance.

4.1.3 Transition caching

One way of exploiting the dependency information in the matrix is by using transition caching. Only the dependent parts of the transition are stored in a cache $\mathcal{C}_k$ for every group k by using the projection function π_k, as described in [4] and shown in Alg. 4.1. This way time is saved, because caching of transitions avoids calling GROUP-NEXT at every step. The density of the matrix has great influence on the performance of caching and of the symbolic tools.

4.2 PBES Language Module

In this section we describe states, transition groups and the dependency matrix for PPGs. We assume to have a rewriter *simplify* that is powerful enough to evaluate any closed data expression to **true** or **false** or to a disjunction or conjunction of predicate variables with closed data expressions as parameters. We use the same rewriter by Van Weerdenburg [19] as used in [16].

Algorithm 4.1 NEXT-CACHE(s, k) computes successors of s for group k using a cache.

NEXT-CACHE(s, k)
1: UPDATE-CACHE(s,k)
2: $S := \emptyset$
3: **for all** $t \in \mathcal{C}_k[\pi_k(s)]$ **do**
4: $\quad t' :=$ NEXT-APPLY(s,t,k)
5: $\quad$ Add t' to S
6: **return** S;

UPDATE-CACHE(s, k)
1: **if** $\pi_k(s) \notin \mathrm{dom}(\mathcal{C}_k)$ **then**
2: $\quad S := \emptyset$
3: $\quad S' :=$ GROUP-NEXT(s,k)
4: $\quad$ **for all** $s' \in S'$ **do**
5: $\quad\quad$ Add $\pi_k(s')$ to S
6: $\quad \mathcal{C}_k[\pi_k(s)] := S$

NEXT-APPLY(s, t, k)
1: $j := 1$
2: **for** $1 \le i \le N$ **do**
3: $\quad$ **if** $D_{k,i} = 0$ **then**
4: $\quad\quad s'[i] := s[i]$
5: $\quad$ **else**
6: $\quad\quad s'[i] := t[j]$
7: $\quad\quad j := j+1$
8: **return** s';

4.2.1 States and transition groups

For PPGs, the state vector is partitioned as follows: $\langle \mathsf{X}, x_1, x_2, \ldots, x_M \rangle$, where X is a propositional variable, and for $i \in \{1 \ldots M\}$ each x_i is the value of parameter i. M is the total number of parameter signatures in the system (consisting of name and type).

We assume the existence of a function *priority* : $S \to \mathsf{Int}$ that assigns a priority to each state (based on the block of the corresponding equation) and a function *player* : $S \to \{\mathbf{0}, \mathbf{1}\}$ that assigns a player to each state (**0** if the corresponding expression is a disjunction, **1** if it is a conjunction). In particular, the **true** state has priority 0 and is owned by player **1** and the **false** state has priority 1 and belongs to player **0**.

The equations in the PPG specify the transitions between states. The right hand side of the equation is split into conjuncts or disjuncts if possible, which form the *transition groups*, which are numbered subsequently. We use a mapping $\mathsf{var} : \mathsf{Int} \to \mathcal{X}$ from group number to variable and a mapping $\mathsf{expr} : \mathsf{Int} \to \mathcal{F}$ from group number to corresponding conjunct or disjunct. In the following we assume the index sets I and J to be disjoint.

For a sequence of equations of the form

$$\sigma \mathsf{X}(\vec{d} : D) = \bigwedge_{i \in I} f_i \wedge \bigwedge_{j \in J} \forall \vec{v} \in D_j \,.\, \big(g_j(\vec{d}, \vec{v}) \Rightarrow \mathsf{X}_j(e_j(\vec{d}, \vec{v}))\big),$$

for each $i \in I$ there is a group k with $\mathsf{expr}(k) = f_i$ and for each $j \in J$ there is a group k with

$$\mathsf{expr}(k) = \forall \vec{v} \in D_j \,.\, \big(g_j(\vec{d}, \vec{v}) \Rightarrow \mathsf{X}_j(e_j(\vec{d}, \vec{v})),$$

and $\mathsf{var}(k) = \mathsf{X}$. Symmetrically for disjunctive equations.

Example 4.3. We will explain these concepts using a specification of two sequential buffers (`buffer.2`):

$$\begin{aligned}
\textbf{proc}\ \mathsf{In}(i : \mathrm{Pos}, q : \mathrm{List}(D)) = \quad & \sum_{d:D} (\#q < 2) \to \mathrm{r}_1(d) \,.\, \mathsf{In}(i, q \triangleleft d) \\
& + (q \neq []) \to \mathrm{w}(i+1, \mathrm{head}(q)) \,.\, \mathsf{In}(i, \mathrm{tail}(q)); \\
\textbf{proc}\ \mathsf{Out}(i : \mathrm{Pos}, q : \mathrm{List}(D)) = & \sum_{d:D} (\#q < 2) \to \mathrm{r}(i, d) \,.\, \mathsf{Out}(i, q \triangleleft d) \\
& + (q \neq []) \to \mathrm{s}_4(\mathrm{head}(q)) \,.\, \mathsf{Out}(i, \mathrm{tail}(q)); \\
\textbf{init}\quad & \textbf{hide}(\{\mathrm{c}\}, \textbf{allow}(\{\mathrm{r}_1, \mathrm{c}, \mathrm{s}_4\}, \textbf{comm}(\{\mathrm{w} \mid \mathrm{r} \to \mathrm{c}\},\ \mathsf{In}(1, []) \parallel \mathsf{Out}(2, [])\ ))));
\end{aligned}$$

The specification of the initial state the system is specified as composed of an In and an Out component, composed with the *parallel composition* ($\parallel$) operator. *Synchronisation* of r and w actions of the two processes proceeds in two steps. The simultaneous occurence of actions r and w (the multi-action $\mathrm{w} \mid \mathrm{r}$) is renamed to c (**comm**) and separate occurances of r and w are ruled out by the *restriction* operator (**allow**). The internal action c is *hidden* (**hide**). This specification is translated to a single process by *linearising* it to Linear Process Specification (LPS) format. The result is the following specification:

$$\begin{aligned}\textbf{proc } \mathsf{P}(q_{in}, q_{out} : \mathrm{List}(D)) = \sum_{d:D} & (\#q_{in} < 2) \to \mathrm{r}_1(d) \,.\, \mathsf{P}(q_{in} \triangleleft d, q_{out}) \\ & + (q_{out} \neq []) \to \mathrm{s}_4(\mathrm{head}(q_{out})) \,.\, \mathsf{P}(q_{in}, \mathrm{tail}(q_{out})); \\ & + (q_{in} \neq [] \wedge \#q_{out} < 2) \to \textbf{tau} \,.\, \mathsf{P}(\mathrm{tail}(q_{in}), q_{out} \triangleleft \mathrm{head}(q_{in})) \\ & + \textbf{delta};\end{aligned}$$

$$\textbf{init} \quad \mathsf{P}([], []);$$

The result of hiding the c action is the internal **tau** transition in the third summand. Actions that are not in the set $\{\mathrm{r}_1, \mathrm{c}, \mathrm{s}_4\}$ are replaced by a **delta** as a result of the restriction operator.
For this process specification, we want to verify the property that if a message is read through r_1, it will eventually be sent through s_4:

$$\nu \mathsf{Y} \,.\, (\forall d : D \,.\, ([\mathrm{r}_1(d)] (\mu \mathsf{X} \,.\, (\langle \textbf{true} \rangle \textbf{true} \wedge [\neg \mathrm{s}_4(d)] \mathsf{X})))) \wedge [\textbf{true}] \mathsf{Y}$$

Satisfaction of this formula by the LPS translates to the following PBES:

$$\begin{aligned}\textbf{pbes } \nu \mathsf{Y}(q_{in}, q_{out} : \mathrm{List}(D)) = & \\ & (\forall_{d:D} \,.\, (\#q_{in} < 2) \Rightarrow \mathsf{X}(q_{in} \triangleleft d, q_{out}, d)) && (1) \\ & \wedge (\forall_{d_0:D} \,.\, (\#q_{in} < 2) \Rightarrow \mathsf{Y}(q_{in} \triangleleft d_0, q_{out})) && (2) \\ & \wedge ((q_{out} \neq []) \Rightarrow \mathsf{Y}(q_{in}, \mathrm{tail}(q_{out}))) && (3) \\ & \wedge ((q_{in} \neq [] \wedge \#q_{out} < 2) \Rightarrow \mathsf{Y}(\mathrm{tail}(q_{in}), q_{out} \triangleleft \mathrm{head}(q_{in}))); && (4) \\ \mu \mathsf{X}(q_{in}, q_{out} : \mathrm{List}(D), d : D) = & \\ & (\#q_{in} < 2) \vee (q_{out} \neq []) \vee (q_{in} \neq [] \wedge \#q_{out} < 2) && (5) \\ & \wedge (\forall_{d_0:D} \,.\, (\#q_{in} < 2) \Rightarrow \mathsf{X}(q_{in} \triangleleft d_0, q_{out}, d)) && (6) \\ & \wedge ((\mathrm{head}(q_{out}) \neq d) \wedge (q_{out} \neq []) \Rightarrow \mathsf{X}(q_{in}, \mathrm{tail}(q_{out}), d)) && (7) \\ & \wedge ((q_{in} \neq [] \wedge \#q_{out} < 2) \Rightarrow \mathsf{X}(\mathrm{tail}(q_{in}), q_{out} \triangleleft \mathrm{head}(q_{in}), d)); && (8) \\ \textbf{init } \mathsf{Y}([], []);\end{aligned}$$

For this equation system, the structure of the state vector is $\langle \mathsf{X}, q_{in}, q_{out}, d \rangle$. The initial state would be encoded as $\langle \mathsf{Y}, [], [], 0 \rangle$. Since the initial state has no parameter d, a default value is chosen. The numbers 1–8 behind the equation parts denote the different transition groups, i.e., each conjunct of a conjunctive expression forms a group. For instance, for group 3 the associated expression is $\mathsf{expr}(3) = ((q_{out} \neq []) \Rightarrow \mathsf{Y}(q_{in}, \mathrm{tail}(q_{out})))$ and it is associated with variable $\mathsf{var}(3) = \mathsf{Y}$. Group 1 encodes the $[\mathrm{r}_1(d)]\varphi$ part of the formula (where φ is the $\mu\mathsf{X}$ part of the formula), groups 2–4 encode the $[\textbf{true}]\mathsf{Y}$ part, group 5 encodes that a transition is enabled ($\langle \textbf{true} \rangle \textbf{true}$), and groups 6–8 encode the cases that not an $\mathrm{r}_4(d)$ transition is taken.

For an equation $\sigma \mathsf{X}(\vec{d} : D) = \varphi$, let $\mathsf{params}(\mathsf{X})$ be the list of parameters $\vec{d}$ and $\mathsf{params}(\mathsf{X})_i$ the i-th element of that list. The next state function GROUP-NEXT is defined as follows. For every k with $\mathsf{var}(k) = \mathsf{X}$,

$$\textsc{Group-Next}(\mathsf{X}(\vec{e}), k) \stackrel{\text{def}}{=} \begin{cases} \{\mathit{simplify}(f[\mathsf{params}(\mathsf{X}) := \vec{e}])\} \\ \qquad \text{if } f = \mathsf{expr}(k) \text{ is a simple formula;} \\ \{\mathsf{X}'(h(\vec{e}, \vec{v})) \mid \vec{v} \in D \wedge g(\vec{e}, \vec{v})\} \\ \qquad \text{if } \mathsf{expr}(k) \text{ is of the form } \mathsf{Q}\vec{v} \in D \,.\, \big(g(\vec{e}, \vec{v}) \oplus \mathsf{X}'(h(\vec{e}, \vec{v}))\end{cases}$$

Note that if f is a simple expression, *simplify*$(f[\mathsf{params}(\mathsf{X}) := \vec{e}])$ will result in either **true** or **false**. In the case that f is not simple, all concrete variable instantiations are enumerated for every quantifier variable $\vec{v}$ for which the guard g is satisfied.

Example 4.4. For the example above, GROUP-NEXT$(\mathsf{Y}([],[]),3)$ yields the empty set because $q_{out} = []$. GROUP-NEXT$(\mathsf{Y}([],[]),2)$ results in $\{\mathsf{Y}([d_1],[]),\mathsf{Y}([d_2],[])\}$.

4.2.2 Dependency matrix

Let $\mathsf{occ}(\varphi)$ be the set of propositional variable occurring in a term φ, let $\mathsf{free}(d)$ be the set of *free data variables* occurring in a data term d, and $\mathsf{used}(\varphi)$ the set of free data variables occurring in an expression φ such that the variables are not merely passed on to the next state. E.g., with $\mathsf{X}(a,b) = \xi$, for the expression $\varphi = a \wedge \mathsf{X}(c,b)$, $\mathsf{used}(\varphi) = \{a,c\}$. Parameter b is not in the set because it does not influence the computation, but is only passed on to the next state. For a formula φ, the function $\mathsf{changed}(\varphi)$ computes the variable parameters changed in the formula:

$$\mathsf{changed}(\mathsf{X}(e_1,\ldots,e_m)) \stackrel{\text{def}}{=} \{d_i \mid i \in \{1\ldots m\} \wedge d_i = \mathsf{params}(\mathsf{X})_i \wedge e_i \neq d_i\}$$

The function $\mathsf{tf}(\varphi)$ determines if φ contains a branch that directly results in a **true** or **false** (not a variable). This is needed because the boolean constants are encoded as a vector with variable names "true" and "false", hence a transition to one of them changes the first part of the state vector. For group k and part i, we define read dependence d_R and write dependence d_W:

$$d_R(k,i) \stackrel{\text{def}}{=} \begin{cases} \textbf{true} & \text{if } i = 1; \\ p_i \in (\mathsf{params}(\mathsf{var}(k)) \cap \mathsf{used}(\mathsf{expr}(k))) & \text{otherwise.} \end{cases}$$

$$d_W(k,i) \stackrel{\text{def}}{=} \begin{cases} (\mathsf{occ}(\mathsf{expr}(k)) \setminus \{\mathsf{var}(k)\} \neq \emptyset) \vee \mathsf{tf}(\mathsf{expr}(k)) & \text{if } i = 1; \\ p_i \in \mathsf{changed}(\mathsf{expr}(k)) & \text{otherwise.} \end{cases}$$

$d_R(k,1)$ is true for every group k, since the variable has to be read to determine if a transition group is applicable.

Definition 4.5 (PPG Dependency matrix)**.** For a PPG P the dependency matrix $DM(P)$ is a $K \times M$ matrix defined for $1 \leq k \leq K$ and $1 \leq i \leq M$ as:

$$DM(P)_{k,i} = \begin{cases} + & \text{if } d_R(k,i) \wedge d_W(k,i); \\ r & \text{if } d_R(k,i) \wedge \neg d_W(k,i); \\ w & \text{if } \neg d_R(k,i) \wedge d_W(k,i); \\ - & \text{otherwise.} \end{cases}$$

Example 4.6. For the PBES in Example 4.3, the dependency matrix looks like this:

k	X	q_{in}	q_{out}	d
1	+	+	−	w
2	+	+	−	−
3	+	−	+	−
4	+	+	+	−
5	+	r	r	−
6	+	+	−	−
7	+	−	+	r
8	+	+	+	−

The first row lists the state vector parts. The left column lists the group numbers. A '+' denotes both read and write dependency, 'w' denotes write dependency, 'r' read dependency, and '$-$' no dependency between the group and the state vector part. For group 1 we can see that the variable is changed from Y to X, which results in a '+' in the X column. The q_{in} parameter is both read and changed (d is added to it). The q_{out} parameter is not touched, which results in a '$-$'. The parameter d is not in $\mathsf{params}(\mathsf{Y})$ and therefore there is no read dependence. However, the value of d is set for the next state, resulting in a 'w' in the last column.

5 Performance Evaluation

In this section we report the performance of our tools compared to existing tools in the mCRL2 toolset.

5.1 Experiment setup

As input we used PBESs that are derived from the following mCRL2 models: n sequential buffers (`buffer-*`), the Sliding Window Protocol (SWP), the IEEE 1394 protocol, a Sokoban puzzle, and state machines that are part of the control system for an experiment at CERN (`wheel_sector`), described in [12]. The models are combined with μ-calculus properties that check absence of deadlock (`nodeadlock`, see Example 2.6), if x is read, then eventually x will be written (`evt_send`, see Example 4.3), or that from the initial state there is a path on which a push action is possible (`always_push`: $\langle \mathbf{true}^* \rangle \langle \mathrm{push} \rangle \mathbf{true}$ – only applicable to the Sokoban puzzle).

As preprocessing steps, we applied `pbesparelm` and `pbesrewr -psimplify` to every equation system, which are rewriters that apply some obvious simplifications to the equation systems. In the reported cases no transformation to PPG was needed, as the systems were already in the required form.

The tools that we compared are:

Tool	Toolset	Groups	Caching	Distributed	Symbolic	Command
`pbes2bes`	mCRL2	–	–	–	–	`pbes2bes -rjittyc`
`pbespgsolve`	mCRL2	–	–	–	–	`pbespgsolve -rjittyc -g`
`pbes2lts -black`	LTSMIN	no	no	no	no	`pbes2lts-grey --black --always-split`
`pbes2lts -grey`	LTSMIN	yes	no	no	no	`pbes2lts-grey --grey --always-split`
`pbes2lts -cache`	LTSMIN	yes	yes	no	no	`pbes2lts-grey -rgs -c --always-split`
`pbes2lts-mpi-*`	LTSMIN	yes	yes	yes	no	`pbes2lts-mpi -rgs -c --always-split`
`pbes-reach`	LTSMIN	yes	no	no	yes	`pbes-reach --order=chain-prev --saturation=sat-like --save-levels -rgs --always-split`

It is indicated whether transition groups, caching, distributed generation or symbolic generation are available. `pbes2bes` and `pbespgsolve` from the mCRL2 toolset are similar in functionality, but different in implementation. For `pbespgsolve` the `-g` option means only generating the parity game without solving. For the LTSMIN tools `pbes2lts-*` and `pbes-reach` the option `-rgs` enables regrouping, `-c` enables caching, and `--black` disables the use of transition groups. `pbes-reach` uses the `sat-like` saturation strategy.

The experiments were performed on a cluster of 10 machines with each two quad-core Intel Xeon E5520 CPUs @ 2.27 GHz (with 2 hyperthreads per core) and 24GB memory. Every tool was given a 20 GB memory limit and a 10 ks time limit. Elapsed time and memory usage have been measured by the tool `memtime`. The experiments were executed using Linux 2.6.34, mCRL2 svn rev. 10785 and for LTSMIN the git rev. after commit 4d11bc20 in the experimental 'next' branch. The tools were built using GCC 4.4.1. Open MPI 1.4.3 was used for the distributed tool.

Table 1: Time performance in seconds. 'T' indicates a timeout, 'M' out of memory.

Equation system	# States	pbes2bes	pbespgsolve	pbes2lts -black	pbes2lts -grey	pbes2lts -cache	pbes2lts-mpi-1	pbes2lts-mpi-4	pbes2lts-mpi-8	pbes-reach
swp.nodeadlock	1,862	5	5	5	5	5	5	5	5	5
swp.evt_send	33,554	7	7	8	11	5	5	5	8	5
1394.nodeadlock	173,101	199	202	231	1,387	120	125	56	73	114
sokoban.372.always_push	834,397	69	78	258	T	403	419	182	62	31
buffer.7.nodeadlock	823,545	32	33	48	76	13	16	9	7	9
buffer.7.evt_send	2,466,257	111	107	157	266	22	27	13	11	9
buffer.8.nodeadlock	5,764,803	235	237	357	594	82	93	31	20	37
buffer.8.evt_send	17,281,283	820	859	1,256	2,171	158	191	71	67	42
buffer.9.nodeadlock	40,353,607	1,059	M	2,937	4,905	571	686	241	197	274
buffer.9.evt_send	121,021,455	M	M	T	T	1,172	1,448	520	306	282
wheel_sector.nodeadlock	4,897,760	T	T	T	T	2,337	2,368	828	939	1,904

Table 2: Memory usage in MB. 'T' indicates a timeout, 'M' out of memory.

Equation system	# States	pbes2bes	pbespgsolve	pbes2lts -black	pbes2lts -grey	pbes2lts -cache	pbes2lts-mpi-1	pbes2lts-mpi-4	pbes2lts-mpi-8	pbes-reach
swp.nodeadlock	1,862	12	11	17	17	16	13	15	14	16
swp.evt_send	33,554	58	29	20	20	18	15	15	16	47
1394.nodeadlock	173,101	227	168	31	30	89	86	60	50	57
sokoban.372.always_push	834,397	1,187	768	34	T	220	217	69	45	47
buffer.7.nodeadlock	823,545	965	354	32	32	91	89	36	27	49
buffer.7.evt_send	2,466,257	3,340	1,215	63	64	181	179	67	43	49
buffer.8.nodeadlock	5,764,803	7,179	2,579	117	117	528	525	145	81	49
buffer.8.evt_send	17,281,283	18,136	9,056	345	345	1,155	1,152	377	204	49
buffer.9.nodeadlock	40,353,607	18,451	M	737	737	4,129	4,127	1,048	538	49
buffer.9.evt_send	121,021,455	M	M	T	T	9,209	9,206	3,003	1,487	49
wheel_sector.nodeadlock	4,897,760	T	T	T	T	1,288	1,285	389	238	90

5.2 Results

Results are in Tables 1 (time performance in seconds) and 2 (memory usage in MB). For the MPI tool, the values are the maximum for the workers. The 'T' indicates a timeout, the 'M' indicates an Out of Memory error. We can make the following observations.

From the results we see that `pbes2bes` and `pbespgsolve` from the mCRL2 toolset perform better than `pbes2lts -black`, the LTSMIN based tool without any optimisation. The memory performance of the LTSMIN tool however is much better, even over 25 times better in the case of `buffer.8.evt_send`.

Looking at `pbes2lts -grey` we observe that only splitting into transition groups without any optimisations has a negative impact on the performance, especially in the case of `1394.nodeadlock`.

The LTSMIN tools have a relatively bad performance for the Sokoban puzzle, because of the structure of `always_push`: either "we can do a push now" or "we move and take a recursive step". If this formula is evaluated as a whole on a state where we can do a push, the first part will immediately evaluate to **true** and the formula as well, without taking the recursive step. When the formula is split into transition groups, then both parts may be evaluated independently. Although the second part is not needed, such on-the-fly solving optimisations are not available in the PBES language module yet when transition groups are enabled. This causes LTSMIN to generate a state space of 10,992,856 states (instead of 834,397), but still the symbolic tool of LTSMIN, `pbes-reach`, is the fastest.

Transition caching pays off for many systems. Compared to the mCRL2 tools, the speedup is between 1.8 and 5.1 for the sequential buffers and for `wheel_sector` the instantiation is completed within the timebound. The distributed tool does not scale well. The speedup with 8 workers compared to 1 worker is 6.8 for the Sokoban puzzle, but does not exceed 4.7 for the sequential buffers, and is only 2.5 for the `wheel_sector` case. In the `wheel_sector` and `1394` cases the execution time for 8 workers is even worse than with 4 workers, indicating that there is a limit to the number of workers that result in a further speedup.

The symbolic tool performs best of all sequential tools in all cases. The tool is up to 19.5 times faster than the fastest tool from the mCRL2 toolset (in the `buffer.8.evt_send` case). And for some cases LTSMIN could finish within memory and time bounds, whereas the mCRL2 tools could not. Memory usage of `pbes-reach` is slightly worse in the smallest cases, but up to more than 180 times better than the mCRL2 tools for the other cases.

6 Conclusions

We have defined PPG as normal form for PBESs and a transformation to PPG, making the instantiation to parity games more straightforward. We implemented a PBES language module for LTSMIN. As a result, the high-performance capabilities for state space generation become available for parity game generation. We demonstrated this for distributed state space generation and for symbolic state space generation.

Experimental comparison to existing tools shows good results. The LTSMIN tools reduce memory usage enormously. Transition caching, distributed computation and the symbolic tool speed up the instantiation in all reported cases. However, the distributed tool does not scale well. For all reported cases, the symbolic LTSMIN tool performed the best, with up to 19 times speedup and up to more than 180 times lower memory usage compared to the mCRL2 tools.

We intend to extend the tool with optimisations, such as on-the-fly minimisation and solving, i.e., while generating the parity game (possibly also distributed). Furthermore, the symbolic tool generates a BDD representation of the parity game, which asks for solvers that can deal with such symbolic parity games similar to the tool by [1].

Acknowledgments. We are grateful to Tim Willemse, Jeroen Keiren and Wieger Wesselink for their support on the mCRL2 toolset.

References

[1] M. Bakera, S. Edelkamp, P. Kissmann, and C.D. Renner. Solving μ-Calculus Parity Games by Symbolic Planning. In *MoChArt 2008*, volume 5348 of *LNCS*. Springer, 2009. doi:10.1007/978-3-642-00431-5_2.

[2] S.C.C. Blom, B. Lisser, J.C. van de Pol, and M. Weber. A Database Approach to Distributed State-Space Generation. *Journal of Logic and Computation*, 21(1), 2009. doi:10.1093/logcom/exp004.

[3] S.C.C Blom and J.C. van de Pol. Symbolic Reachability for Process Algebras with Recursive Data Types. In *ICTAC 2008*, volume 5160 of *LNCS*. Springer, 2008. doi:10.1007/978-3-540-85762-4_6.

[4] S.C.C. Blom, J.C. van de Pol, and M. Weber. Bridging the Gap between Enumerative and Symbolic Model Checkers. Technical Report TR-CTIT-09-30, CTIT, University of Twente, Enschede, 2009.

[5] S.C.C. Blom, J.C. van de Pol, and M. Weber. LTSmin: Distributed and Symbolic Reachability. In *CAV 2010*, volume 6174 of *LNCS*. Springer, 2010. doi:10.1007/978-3-642-14295-6_31.

[6] J.C. Bradfield and C. Stirling. Modal logics and mu-calculi: An introduction. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 4, pages 293–330. Elsevier, 2001.

[7] T. Chen, B. Ploeger, J.C. van de Pol, and T.A.C. Willemse. Equivalence Checking for Infinite Systems Using Parameterized Boolean Equation Systems. In *CONCUR 2007*, volume 4703 of *LNCS*. Springer, 2007. doi:10.1007/978-3-540-74407-8_9.

[8] A. van Dam, B. Ploeger, and T.A.C. Willemse. Instantiation for Parameterised Boolean Equation Systems. In *ICTAC 2008*, volume 5160 of *LNCS*. Springer, 2008. doi:10.1007/978-3-540-85762-4_30.

[9] O. Friedmann and M. Lange. Solving Parity Games in Practice. In *ATVA 2009*, volume 5799 of *LNCS*. Springer, 2009. doi:10.1007/978-3-642-04761-9_15.

[10] J.F. Groote and T.A.C. Willemse. Model-checking processes with data. *Science of Computer Programming*, 56(3), 2005. doi:10.1016/j.scico.2004.08.002.

[11] J.F. Groote and T.A.C. Willemse. Parameterised boolean equation systems. *Theoretical Computer Science*, 343(3), 2005. doi:10.1016/j.tcs.2005.06.016.

[12] Y.-L. Hwong, V.J.J. Kusters, and T.A.C. Willemse. Analysing the Control Software of the Compact Muon Solenoid Experiment at the Large Hadron Collider. ArXiv:1101.5324v1, 2011.

[13] M. Jurdziński. Small Progress Measures for Solving Parity Games. In *STACS 2000*, volume 1770 of *LNCS*, 2000. doi:10.1007/3-540-46541-3_24.

[14] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3), 1983. doi:10.1016/0304-3975(82)90125-6.

[15] R. Mazala. Infinite Games. In *Automata Logics, and Infinite Games*, volume 2500 of *LNCS*, pages 197–204. Springer, 2002. doi:10.1007/3-540-36387-4_2.

[16] B. Ploeger, J.W. Wesselink, and T.A.C. Willemse. Verification of reactive systems via instantiation of Parameterised Boolean Equation Systems. *Information and Computation*, 209(4), 2011. doi:10.1016/j.ic.2010.11.025.

[17] J.C. van de Pol and M. Timmer. State Space Reduction of Linear Processes Using Control Flow Reconstruction. In *ATVA 2009*, volume 5799 of *LNCS*. Springer, 2009. doi:10.1007/978-3-642-04761-9_5.

[18] J.C. van de Pol and M. Weber. A Multi-Core Solver for Parity Games. *Electronic Notes in Theoretical Computer Science*, 220(2), 2008. doi:10.1016/j.entcs.2008.11.011.

[19] M.J. van Weerdenburg. *Efficient Rewriting Techniques*. PhD thesis, Eindhoven University of Technology, 2009.

[20] W. Zielonka. Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees. *Theoretical Computer Science*, 200(1–2), 1998. doi:10.1016/S0304-3975(98)00009-7.